

Estructuras de datos

Tarea 2

Federico Terán

Maria Camila

Cacedo



```

1) void algoritmo1(int n){
2   int i, j = 1; → 2
3   for(i = n * n; i > 0; i = i / 2){ → (2 log2n) + 2
4     int suma = i + j; (2 log2n) + 1
5     printf("Suma %d\n", suma); (2 log2n) + 1
6     ++j; (2 log2n) + 1
}
}

```

$$2 + 2 \log_2 n + 2 + 3(2 \log_2 n + 1)$$

$$2 + 2 \log_2 n + 2 + 6 \log_2 n + 3$$

$$\underline{8 \log_2 n + 7}$$

$O(\log_2 n)$

En la linea 2 se puede ver que i empieza en n^2 e itera hasta 0, dividiéndose entre 2 K veces.

$$\frac{n^2}{2^K} = 1 \rightarrow n^2 = 2^K \rightarrow \log_2 n^2 = \log_2 2^K \rightarrow 2 \log_2 n = K$$

Sin embargo, esta formula no cuenta las dos últimas iteraciones, cuando $i=1$ y cuando $i=0$. Por lo tanto, se le debe sumar 2 a la fórmula anterior.

En las líneas siguientes solo se le suma 1 pues en $i=0$ no se cumple la condición por lo que no entra al ciclo.

1.b) Al ejecutarse el algoritmo 1 se imprime una sucesión de números que inicia en $i+j$ siendo $i=n^2$ es decir, $64+1=65$. Posteriormente j va incrementando en 1 mientras i va reduciendo su valor a la mitad, esto se repite 7 veces mientras que i es distinto de 0 mostrando los valores 65, 34, 19, 12, 9, 8, 8

```

2. int algoritmo2(int n){
3.     int res = 1, i, j;      3
4.     for(i = 1; i <= 2 * n; i += 4) n/2 + 1
5.         for(j = 1; j * j <= n; j++)  $\frac{\sqrt{n} + n}{2}$ 
6.             res += 2;  $\frac{\sqrt{n^3} + n^2}{2}$ 
7.     return res; 1 2
}

```

$$\left. \begin{array}{l} \\ \\ \end{array} \right\} O(n^{3/2})$$

En la linea 3 se puede evidenciar que i empieza en 1 y finaliza en $2n$, ademas i aumenta sumandole 4, por lo que se puede traducir a $\frac{2n}{4}$ y posteriormente simplificar a $\frac{n}{2}$.

Finalmente se puede concluir que la linea se ejecuta $\frac{n}{2} + 1$ ya que la linea se ejecuta una ultima vez mas en la cual no ingresa al ciclo.

Las ejecuciones de la linea 4 se pueden deducir a traves de probar con valores de n

$$n = 5$$

$$n = 16$$

$$n = 64$$

$$\text{for } j = 1 2 3$$

$$\text{for } j = 1 2 3 4 5$$

$$\text{for } j = 1 2 3 4 5 6 7 8 9$$

$$\sqrt{5} = 2$$

$$\sqrt{16} = 4$$

$$\sqrt{64} = 8$$

(numero entero)

La condicion que se debe cumplir para que el ciclo itere es que $j^2 < n$, se concluye que la linea se ejecuta \sqrt{n} mas una ultima vez en la que la condicion no se cumple

Línea 4

$$\frac{n}{2}(\sqrt{n}+1) \Rightarrow \frac{n(\sqrt{n}+1)}{2} \Rightarrow \frac{\sqrt{n \cdot n^2} + n}{2} = \frac{\sqrt{n^3} + n}{2}$$

Línea 5

$$\frac{n}{2}(\sqrt{n}) \Rightarrow \frac{n\sqrt{n}}{2} \Rightarrow \frac{\sqrt{n \cdot n^2}}{2} \Rightarrow \frac{\sqrt{n^3}}{2}$$

$$3 + \left(\frac{n}{2} + 1\right) + \left(\frac{\sqrt{n^3} + n}{2}\right) + \frac{\sqrt{n^3}}{2} + 1$$

$$5 + 2 \frac{n}{2} + \frac{\sqrt{n^3}}{2} = \frac{\sqrt{n^3}}{2} + 5 = \frac{n^{3/2}}{2} + 5$$

- 2b) Al ejecutar el algoritmo 2, sabemos que
 $n=8$, j solo se va a ejecutar 2 veces por cada iteración de i, pues j se ejecuta mientras $j^2 \leq n$, y como en la tercera iteración j^2 sería = 9, y $9 > 8$, no se le sumaría más a la variable res. Sabiendo esto, y que i se ejecuta 4 veces, a la variable $res = 1$ se le sumará $2 \times 2 \times 4$, es decir 16 en las diferentes iteraciones por lo que res termina siendo 17.

```

1 void algoritmo3(int n){
2     int i, j, k; 3
3     for(i = n; i > 1; i--) n
4         for(j = 1; j <= n; j++) n2 - 1 →  $\sum_{i=1}^{n-1} i+1$ 
5             for(k = 1; k <= i; k++)
6                 printf("Vida cruel!!\n");
}

```

$$3 + n + n^2 - 1 + n \cdot \sum_{i=1}^{n-1} i + 1 + n \cdot \sum_{i=1}^{n-1} i$$

$$2 + n + n^2 + n \cdot \sum_{i=1}^{n-1} 1 + n \cdot \sum_{i=1}^{n-1} i + n \cdot \sum_{i=1}^{n-1} i$$

$$2 + n + n^2 + n \cdot (n-1) + 2 \left[n \cdot \left(\frac{(n-1) \cdot n}{2} \right) \right]$$

$$2 + n + n^2 + n^2 - n + 2 \cdot \left[n \cdot \frac{n^2 - n}{2} \right]$$

$$2 + 2n^2 + 2 \cdot \frac{n^3 - n^2}{2}$$

$$2 + 2n^2 + n^3 - n^2$$

$$n^3 + 2n^2 - 2n + 2 \quad n=3$$

① 3

2

1

① 1 2 3 4

1 2 3 4

④ 1234 1234 1234
111 111 111

123 123 123
11 11 11

en este ejemplo, cuando $n=3$, se observa que K se repite, por cada i, 3 veces, es decir n veces y todo esto se repite $n-1$ veces, es decir que las veces que se itera K es una sumatoria de i hasta $n-1$ de i+1 n veces, y el print se ejecuta 1 vez menos que K

```

int algoritmo4(int* valores, int n) {
    int suma = 0, contador = 0; 2
    int i, j, h, flag; 4
    for(i = 0; i < n; i++){ n+1
        j = i + 1; n
        flag = 0; n
        while(j < n && flag == 0){
            if(valores[i] < valores[j]){
                for(h = j; h < n; h++){
                    suma += valores[i];
                }
            }
        else{ 0
            contador++; 0
            flag = 1; 0
        }
        ++j; →  $\sum_{i=1}^{n-1} i$ 
    }
    return contador;
}

```

Peor	Mejor
$\sum_{i=1}^n i$	$2n$
$\sum_{i=1}^{n-1} i$	n
$\sum_{i=1}^{n-1} \sum_{h=i+1}^{n-1} j+1$	0
$\sum_{i=1}^{n-1} \sum_{j=1}^{n-i} j$	0
	n
	n
	n
	n
	1

Peor caso

En el algoritmo 4, el ciclo de i que va desde 0 hasta n , se repite n veces más la vez en la que no se cumple la condición, es decir, $n+1$ veces, la parte de código que le sigue se repite una vez menos hasta el ciclo de j . En el ciclo de j nos damos cuenta que j se repite primero n veces, luego $n-1$ veces y así hasta llegar a 1, lo que resulta en una sumatoria desde 1 hasta n luego, el if se repite también la misma cantidad menos 1, que es cuando ya no se cumple la condición. El for que viene después, se ejecuta la misma cantidad que j , solo que como está anidado, también se repite las mismas veces que el for más las veces que se repite k si fuera un único ciclo, lo que termina en dos sumatorias anidadas. La suma se ejecuta esa misma cantidad pero en vez de ejecutarse $\sum_{j=1}^{n-1} j+1$, solo se ejecuta con j (por las veces $j=1$ que entra en el ciclo), el else no se ejecuta, y por último, $j++$ se ejecuta las mismas veces que el if porque solo está en el ciclo de j y no en el de $*.$ $O(n^3) \rightarrow$ Tres ciclos anidados

Mejor caso

En el mejor caso, la primera parte se repite igual hasta el `while`, que se repite solo dos veces por cada `i`, es decir, $2n$, una, cuando se cumplen las 2 condiciones y otra cuando ya solo se cumple 1 y no ejecuta lo que hay adentro. Luego, el `if`, que se repite una vez por `i` para verificar que no se cumple la condición y el resto de ejecuciones que se realizan en `else`, que se repiten la misma cantidad de veces, es decir, n en total.

$$O(n^2) \rightarrow \text{solo 2 aciertos anidados}$$

```
void algoritmo5(int n){  
    int i = 0; 1  
    while(i <= n){ 7  
        printf("%d\n", i); 6  
        i += n / 5; 6  
    }  
}
```

O(1)

la complejidad de este ejercicio es constante
Pues, a pesar de que hay algunos casos
de números pequeños en los que el
while se ejecuta más de 7 veces, entre
el número se vuelve más grande, las
repeticiones se vuelven constantes.
Hay que tener en cuenta que si n es
menor que 5 el ciclo sería infinito.

6.

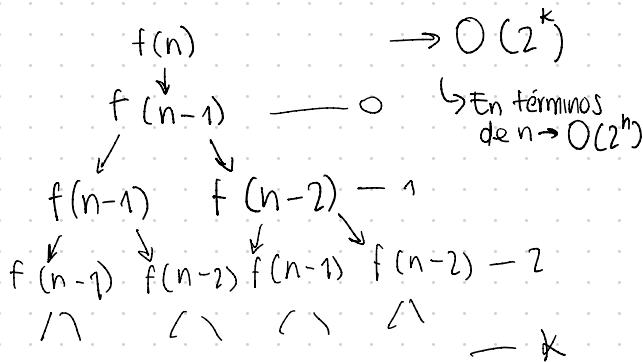
Tamaño Entrada	Tiempo	Tamaño Entrada	Tiempo
5	0,063 s	35	2,4 s
10	0,064 s	40	26,5 s
15	0,080 s	45	5m 33 s
20	0,080 s	50	62 m
25	0,045 s	60	
30	0,270 s	100	

Mucho
Tiempo

```

1 def fiboRecursiva(n):
2     if n < 2:
3         return n
4     else:
5         return fiboRecursiva(n-1) + fiboRecursiva(n-2)
6
7 print(fiboRecursiva(9))

```



La función Fibonacci definida recursivamente es una función con un costo computacional suamente grande,

el tiempo de procesamiento aumentaba rápidamente a medida que se aumentaba la variable de entrada n , el valor

mas alto que se pudo procesar fue $n = 50$, sin embargo el tiempo que se demora es de 62 minutos, lo que es demasiado

para un algoritmo. Teniendo en cuenta que para calcular la función $fibo(n)$ recurre a $fibo(n-1) + fibo(n+2)$, se puede evidenciar que cada vez que se invoca a la función esta recurre 2 veces a la misma función, y a su vez

cada una de las 2 funciones recurrentes recurre cada una a la función

Este comportamiento se puede asociar al de un árbol binario como el siguiente

Se deduce que la complejidad computacional de la función es $O(2^n)$ lo que explicaría el crecimiento tan rápido de
del tiempo

7.

Tamaño Entrada	Tiempo	Tamaño Entrada	Tiempo
5	0,064 s	45	0,064 s
10	0,065 s	50	0,065 s
15	0,064 s	100	0,065 s
20	0,062 s	200	0,062 s
25	0,064 s	500	0,063 s
30	0,064 s	1000	0,062 s
35	0,063 s	5000	0,064 s
40	0,063 s	10000	0,064 s

```

1 def fiboCiclos(n):
2     ans = 0      ↴
3     a = 0      ↴
4     b = 1      ↴
5     for i in range(n):    n + 2
6         if i % 2 == 0:
7             ans = b + a
8             b = ans
9
10        else:
11            ans = a + b
12            a = ans
13
14    return ans      ↴

```

$$1 + 1 + 1 + (n+1) + 3n + 1 = 4n + 5$$

$\mathcal{O}(n)$

8. Ejecute la operación mostrarPrimos que presento en su solución al ejercicio 4 de la Tarea 1 y también la versión de la solución a este ejercicio que se subirá a la página del curso con los siguientes valores y mida el tiempo de ejecución:

Tamaño Entrada	Tiempo Solución Propia	Tiempo Solución Profesores
100		
1000	0,110 s	0,110 s
5000	0,110 s	0,110 s
10000	0,120 s	0,110 s
50000	0,342 s	0,120 s
100000	5,159 s	0,223 s
200000	10,7 s	0,380 s
	1m 15 s	0,850 s

Responda las siguientes preguntas:

- (a) ¿qué tan diferentes son los tiempos de ejecución y a qué cree que se deba dicha diferencia?
(b) ¿cuál es la complejidad de la operación o bloque de código en el que se determina si un número es primo en cada una de las soluciones?

a)

Los algoritmos difieren en los tiempos debido a que a en nuestro algoritmo se hace más trabajo del necesario. Pues en nuestro caso para encontrar un primo se recorren todos los numero desde 2 hasta n cuando no es necesario. Esto se puede evidenciar en el código que proveen los profesores que solo se ejecuta desde 2 hasta raíz de n y así con muchos otros casos.

B.

```
def esPrimo(n):
    if n < 2: ans = False
    else:
        i, ans = 2, True
        while i * i <= n and ans:
            if n % i == 0: ans = False
            i += 1
    return ans
```

```
def mostrarPrimos (numero):
    primos=[]
    sumaPrimos=[]
    for i in range (1,numero):
        n=i
        suma=0
        variable=calcularPrimos(i)
        if variable== True:
            primos.append(i)
            while n != 0:
                suma+= n%10
                n=n//10
            variableSuma=calcularPrimos(suma)
            if variableSuma == True:
                sumaPrimos.append(i)
        for i in range (0,len(primos)):
            if i < len(primos)-1:
                print(" -> "+str(primos[i])+",")
            elif i == len(primos)-1:
                print(" -> "+str(primos[i]))
        for i in range (0,len(sumaPrimos)):
            sumaPrimos[i]=str(sumaPrimos[i])
        print()
    print("Números entre 1 y",numero, "con suma de dígitos con valor primo:")
    print(" -> ",join(sumaPrimos))
```

En la solución dada por los profesores, el bloque de código con mayor costo computacional es $O(\sqrt{n})$.

Mientras que en nuestra solución la complejidad es

$$O(\log n)$$